Shapes and Flattening

John Reppy
Joe Wingerter

University of Chicago

March 2020

NESL

- NESL is a first-order functional language for parallel programming over sequences designed by Guy Blelloch [Blelloch '96].
- Provides parallel for-each operation (with optional filter)

```
{ x + y : x in xs; y in ys }
{ x / y : x in xs; y in ys | (y /= 0) }
```

- Provides other parallel operations on sequences, such as reductions, prefix-scans, and permutations.

```
function dot (xs, ys) = sum ({ x * y : x in xs; y in ys })
```

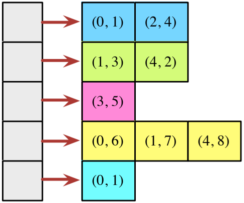
- Supports Nested Data Parallelism (NDP) — components of a parallel computation may themselves be parallel.

NDP example: sparse matrix times dense vector

$$\begin{bmatrix} \mathbf{1} & 0 & \mathbf{4} & 0 & 0 \\ 0 & \mathbf{3} & 0 & 0 & \mathbf{2} \\ 0 & 0 & 0 & \mathbf{5} & 0 \\ \mathbf{6} & \mathbf{7} & 0 & 0 & \mathbf{8} \\ 0 & 0 & \mathbf{9} & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

Want to avoid computing products where matrix entries are 0.

Sparse representation tracks non-zero entries using sequence of sequences of index-value pairs:



NDP example: sparse-matrix times vector

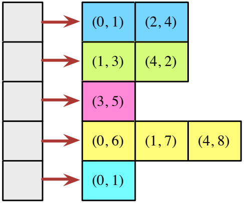
In NESL, this algorithm has a compact expression:

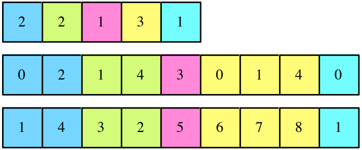
```
function svxv (sv, v) = sum ({ x * v[i] : (i, x) in sv })

function smxv (sm, v) = { svxv (sv, v) : sv in sm }
```

Notice that the `smxv` is a map of map-reduce subcomputations; *i.e.*, nested data parallelism.

NDP example: sparse-matrix times vector

Naive parallel decomposition will be unbalanced because of irregularity in sub-problem sizes.





Flattening transformation converts NDP to flat DP (including AoS to SoA)

Flattening

Flattening (*a.k.a. vectorization*) is a global program transformation that converts irregular nested data parallel code into regular flat data parallel code.

- Lifts scalar operations to work on sequences of values
- Flattens nested sequences paired with segment descriptors
- Conditionals are encoded as data
- Residual program contains vector operations plus sequential control flow and recursion/iteration.

Flattening example: factorial

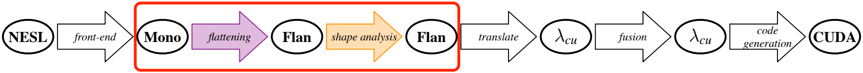
```
function fact (n) = if (n <= 0) then 1 else n * fact(n-1);

function fact↑ (ns) = { fact (n) : n in ns };
```

$\Longrightarrow$

```
function fact↑ (ns) =
  let fs = (ns <=↑ dist(0, #ns));
      (ns1, ns2) = PARTITION(ns, fs);
      vs1 = dist(1, #ns1);
      vs2 = if (#ns2 = 0)
          then []
          else let
            es = (ns2 -↑ dist(1, #ns2));
            rs = fact↑ (es);
            in (ns2 *↑ rs);
      in COMBINE(vs1, vs2, fs)
```

Nessie

- NESL to CUDA compiler built from scratch.
- Front-end produces **Mono**, a monomorphic ANF-style IR.
- Flattening eliminates NDP and produces **Flan**, which is a flat-vector language.
- Shape analysis is used to tag vectors with size information (symbolic in some cases).
- Back-end IR (λ_{cu}) is based on Second-Order Array Combinators (SOAC).
- Back-end maps flattened code to kernels, performs aggressive fusion, compile-time memory management, and CUDA code generation.
- The focus of this work is on the flattening and shape analysis.



Simple map-reduce fusion

The λ_{cu} code for the `dotp` example is

```
kernel prod (xs : [float], ys : [float]) -> [float] {
  let res = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  return res
}
kernel sum (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}

function dotp (xs : [float], ys : [float]) -> [float] {
  let t1 : [float] = run prod (xs, ys)
  let t2 : float = run sum (t1)
  return t2
}
```

Simple map-reduce fusion

Step 1: Fuse the two kernels into a combined kernel.

```
kernel prod (xs : [float], ys : [float]) -> [float] {
  let res = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  return res
}
kernel sum (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}

function dotp (xs : [float], ys : [float]) -> [float] {
  let t1 : [float] = run prod (xs, ys)
  let t2 : float = run sum (t1)
  return t2
}
```

Simple map-reduce fusion

Step 1: Fuse the two kernels into a combined kernel.

```
kernel F (xs : [float], ys : [float]) -> float {
  let ts = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  let res = REDUCE { i => ts[i] using ts } (FADD) (#ts)
  return res
}

function dotp (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Simple map-reduce fusion

Step 2: Fuse the **MAP** operation into the **REDUCE**'s pull operation

```
kernel F (xs : [float], ys : [float]) -> float {
  let ts = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  let res = REDUCE { i => ts[i] using ts } (FADD) (#ts)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Simple map-reduce fusion

Step 2: Fuse the **MAP** operation into the **REDUCE**'s pull operation

```
kernel F (xs : [float], ys : [float]) -> float {
  let res = REDUCE { i => xs[i] * ys[i] using xs, ys } (FADD) (#xs)
  return res
}

function dotp (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Fancier fusion

Consider the following NESL function:

```
function norm2 (xys) : ([float, float]) -> ([float], [float]) =
  let xs = { x : (x, y) in xys };
      ys = { y : (x, y) in xys };
      sum1 = sum(xs);
      gts = { y : y in ys | (y > 0) };
      sum2 = sum(gts);
  in
    ({ x / sum1 : x in xs }, { y / sum2 : y in ys })
```

Fancier fusion *(continued ...)*

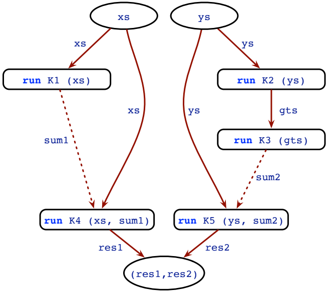
Translating to λ_{cu} produces the following code:

```
kernel K1 (xs : [float]) -> float {
  REDUCE { i => xs[i] using xs } (FADD) (#xs)
}
kernel K2 (ys : [float]) -> [float] {
  FILTER { i => ys[i] using ys } { y => y > 0 } (#ys)
}
kernel K3 (gts : [float]) -> float {
  REDUCE { i => gts[i] using gts } (FADD) (#gts)
}
kernel K4 (xs : [float], s : float) -> [float] {
  MAP { i => xs[i] / s using xs } (#xs)
}
kernel K5 (ys : [float], s : float) -> [float] {
  MAP { i => ys[i] / s using ys } (#ys)
}

function norm2 (xs : [float], ys : [float]) -> ([float], [float])
{
  let sum1 : float = run K1 (xs)
  let gts : [float] = run K2 (ys)
  let sum2 = run K3 (gts)
  let res1 : [float] = run K4 (xs, sum1)
  let res2 : [float] = run K5 (ys, sum2)
  return (res1, res2)
}
```

Fancier fusion *(continued ...)*

PDG control region

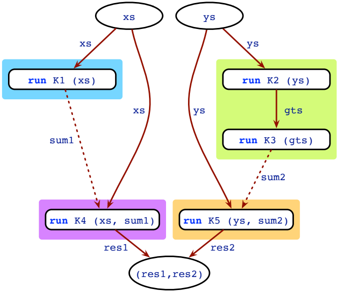
```
kernel K1 (xs : [float]) -> float {
  REDUCE { i => xs[i] using xs } (FADD) (#xs)
}
kernel K2 (ys : [float]) -> [float] {
  FILTER { i => ys[i] using ys } { y => y > 0 } (#ys)
}
kernel K3 (gts : [float]) -> float {
  REDUCE { i => gts[i] using gts } (FADD) (#gts)
}
kernel K4 (xs : [float], s : float) -> [float] {
  MAP { i => xs[i] / s using xs } (#xs)
}
kernel K5 (ys : [float], s : float) -> [float] {
  MAP { i => ys[i] / s using ys } (#ys)
}

function norm2 (xs : [float], ys : [float]) -> ([float], [float])
{
  let sum1 : float = run K1 (xs)
  let gts : [float] = run K2 (ys)
  let sum2 = run K3 (gts)
  let res1 : [float] = run K4 (xs, sum1)
  let res2 : [float] = run K5 (ys, sum2)
  return (res1, res2)
}
```



Fancier fusion *(continued ...)*

The best schedule if $|\texttt{xs}| \stackrel{?}{=} |\texttt{ys}|$ is unknown

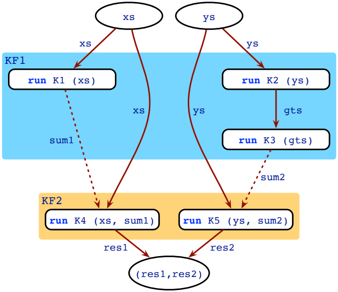
```
kernel K1 (xs : [float]) -> float {
  REDUCE { i => xs[i] using xs } (FADD) (#xs)
}
kernel K23 (gts : [float]) -> float {
  REDUCE { i => if ys[i] > 0 then ys[i] else 0 using gts } (FADD) (#gts)
}
kernel K4 (xs : [float], s : float) -> [float] {
  MAP { i => xs[i] / s using xs } (#xs)
}
kernel K5 (ys : [float], s : float) -> [float] {
  MAP { i => ys[i] / s using ys } (#ys)
}

function norm2 (xs : [float], ys : [float]) -> ([float], [float])
{
  let sum1 : float = run K1 (xs)
  let sum2 : [float] = run K23 (ys)
  let res1 : [float] = run K4 (xs, sum1)
  let res2 : [float] = run K5 (ys, sum2)
  return (res1, res2)
}
```



Fancier fusion *(continued ...)*

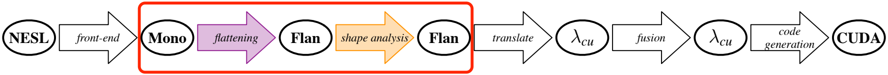
A better schedule, if we know that $|\mathtt{xs}| = |\mathtt{ys}|$

```
kernel K123 (xs : [float], ys : [float]) -> (float, float) {
  REDUCE { i => (xs[i], if ys[i] > 0 then ys[i] else 0)
           using xs, ys }
    (FADD, FADD) (#xs)
}
kernel K45 (xs, sum1, ys, sum2) -> [float] {
  MAP { i => (xs[i] / sum1, ys[1] / sum2) using xs, ys } (#xs)
}

function norm2 (xs : [float], ys : [float]) -> ([float], [float])
{
  let (sum1, sum2) = run K123 (xs, ys)
  let (res1, res2) = run K45 (xs, sum1, ys, sum2)
  return (res1, res2)
}
```



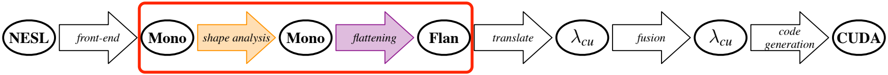
Shape analysis

- Shape analysis should identify when two sequences have the same size.
- It might also detect hyper-rectangular shapes (*e.g.*, dense matrices).
- Examples like the `norm2` function are hard to analyze post-flattening.
- The thesis of this work is that it is better to do the shape analysis before flattening.
- We do shape analysis first on the **Mono** representation and record the information using annotated types.



Shape analysis

- Shape analysis should identify when two sequences have the same size.
- It might also detect hyper-rectangular shapes (*e.g.*, dense matrices).
- Examples like the `norm2` function are hard to analyze post-flattening.
- The thesis of this work is that it is better to do the shape analysis before flattening.
- We do shape analysis first on the **Mono** representation and record the information using annotated types.



Annotated types

We define the following representation for shape information:

$$
\begin{array}{lcll}
\upsilon & ::= & d & \text{fixed dimension} \\
 & | & \phi & \text{dimension function} \\
d & ::= & n & \text{known size} \\
 & | & \alpha & \text{dimension variable} \\
 & | & \phi(\alpha) & \text{applied function} \\
 & | & d_1 + d_2 & \text{dimension addition} \\
 & | & \sum_{\alpha=1}^{d} \phi(\alpha) & \text{summation}
\end{array}
$$

And we annotate types with dimension information.

$$
\begin{array}{lcll}
\widehat{\tau} & ::= & [\widehat{\tau} \ \# \ \upsilon] & \text{sequence type} \\
 & | & \widehat{\tau}_1 \times \widehat{\tau}_2 & \text{pair type} \\
 & | & \widehat{\iota} & \text{base types} \\
\widehat{\iota} & ::= & \mathbf{int}(\upsilon) & \text{integer type} \\
 & | & \cdots & \text{other base types}
\end{array}
$$

We use dimension functions (ϕ) to represent the sizes of irregular nested arrays.
And we annotate the integer type with a dimension to allow tracking of sizes through length computations.

Annotated function types

For builtin operators and user functions we use annotated types with the following general syntax:

$$\forall \vec{\alpha}, \vec{\phi}.(\widehat{\tau}_1, \ldots, \widehat{\tau}_k) \rightarrow \exists \vec{\beta}, \vec{\psi}.\widehat{\tau}$$

which captures the fact that the function can be polymorphic in the shape of its arguments, but the the shape of its result might be unknown (*e.g.*, because of a filter).

Some builtin-function types

$$
\begin{array}{rcl}
+_{\mathbf{int}} & : & \forall\alpha,\beta.(\mathbf{int}(\alpha),\mathbf{int}(\beta)) \to \mathbf{int}(\alpha+\beta) \\
*_{\mathbf{int}} & : & \forall\alpha,\beta.(\mathbf{int}(\alpha),\mathbf{int}(\beta)) \to \exists\gamma.\mathbf{int}(\gamma) \\
\texttt{length} & : & \forall\alpha.([\widehat{\tau}\,\#\,\alpha]) \to \mathbf{int}(\alpha) \\
\texttt{lengths} & : & \forall\alpha,\phi.([[\widehat{\tau}\,\#\,\phi]\,\#\,\alpha]) \to [\mathbf{int}(\phi)\,\#\,\alpha] \\
\texttt{iota} & : & \forall\alpha.(\mathbf{int}(\alpha)) \to \exists\phi.[\mathbf{int}(\phi)\,\#\,\alpha] \\
\texttt{++} & : & \forall\alpha,\beta.([\widehat{\tau}\,\#\,\alpha],[\widehat{\tau}\,\#\,\beta]) \to [\widehat{\tau}\,\#\,\alpha+\beta] \\
\texttt{concat} & : & \forall\alpha,\phi.([[\widehat{\tau}\,\#\,\phi]\,\#\,\alpha]) \to [\widehat{\tau}\,\#\,\sum_{\beta=1}^{\alpha}\phi(\beta)] \\
\texttt{sum}_{\mathbf{float}} & : & \forall\alpha.([\mathbf{float}\,\#\,\alpha]) \to \mathbf{float} \\
\texttt{filter} & : & \forall\alpha.([\widehat{\tau}\,\#\,\alpha],[\mathbf{bool}\,\#\,\alpha]) \to \exists\beta.[\widehat{\tau}\,\#\,\beta]
\end{array}
$$

Examples

Consider the following NESL function that does element-wise multiplication of two nested sequences:

```
function mm (xss:[[float]], yss:[[float]]) =
  { { x + y : x in xs; y in ys }
    : xs in xss, ys in yss };
```

What must be true about the shapes of `xss`, `yss`, and its result?

`xss` and `yss` are irregular, but they must have the same shape, which is also the shape of the result.

$$\mathtt{mm} : \forall\alpha,\phi.([[\textbf{float}\ \#\ \phi]\ \#\ \alpha],\ [[\textbf{float}\ \#\ \phi]\ \#\ \alpha]) \rightarrow [[\textbf{float}\ \#\ \phi]\ \#\ \alpha]$$

Examples

Another example that computes the product of a nested sequence and another sequence:

```
function mv (xss: [[float]], ys: [float]) =
  { { x + y : x in xs; y in ys }
    : xs in xss };
```

What must be true about the shapes of `xss`, `yss`, and its result?

Each row of `xss` must have the same length as `ys` and, thus we infer that `xss` must have rectangular shape.

$$\texttt{mv} : \forall\alpha, \beta.([[\textbf{float} \# \beta] \# \alpha], [\textbf{float} \# \beta]) \to [[\textbf{float} \# \beta] \# \alpha]$$

Shape analysis

- We start with a pre-lifting pass that defines lifted versions of functions that are used in parallel contexts.
- Shape analysis works by introducing equality constraints between shape and dimension expressions.
- Shape constraints are handled symbolically; we do minimal arithmetic reasoning.
- Constraints of the form $\phi = d$ or $\phi(\alpha) = d$ imply that ϕ is a constant dimension function.
- Because shape analysis is done before flattening, we do not lose information about sequences of tuples.

Shape-preserving flattening

- Once we have annotated the **Mono** representation, we apply the flattening transformation.
- In the resulting program, size and segment descriptors are used to specify the iteration space of the parallel SOACs.
- If two descriptors have the same type in the flattened program, then the iteration spaces described by them must be the same and fusion may be possible.

Status

- The shape analysis is implemented in the Nessie compiler.
- We are in the process of implementing the shape-preserving flattening and **Flan** to λ_{cu} translation.
- The main challenge is dealing with the large library of data-parallel operations provided by NESL.

Questions?